NAWCWD TP 8786

Two-Dimensional Distributed Velocity Collision Avoidance

by
Josh L. Wilkerson
Jim Bobinchak
Michael Culp
Josh Clark
Tyler Halpin-Chan
Katia Estabridis
Gary Hewer
Physics Division
Research and Intelligence Department

FEBRUARY 2014

NAVAL AIR WARFARE CENTER WEAPONS DIVISION
CHINA LAKE, CA 93555-6100

Naval Air Warfare Center Weapons Division

FOREWORD

This report summarizes some of the authors' research as part of an effort to understand and assess collision avoidance algorithms both in simulation and in a hardware-in-the-loop (HWIL) arena studies. The Navy's capacity and capability can be expanded and enhanced by integrating autonomous unmanned aerial vehicles (UAVs) into the force. They can both reduce labor intensive mission requirements and minimize many of the risks associated with personnel in potentially adversarial environments. In addition, they offer unique capabilities for intelligence, surveillance, and reconnaissance (ISR) type missions. Collision avoidance is certainly an essential capability for any cooperative venture. This work was done with support from Public Law Section 219 funds during calendar year 2013.

This report was reviewed for technical accuracy by Larry Peterson.

Approved by
L. MERWIN, *Head*
Research and Intelligence Department
11 February 2014

Under authority of
M. T. MORAN
RDML, U.S. Navy
Commander

Released for publication by
S. O'NEIL
Director for Research and Engineering

NAWCWD Technical Publication 8786

Published by ..... Technical Communication Office
Collation ..... Cover, 18 leaves
First printing ..... 20 paper, 21 electronic media

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED (From - To)
11-02-2014	Final	1 January – 31 December 2013

4. TITLE AND SUBTITLE
Two-Dimensional Distributed Velocity Collision Avoidance (U)

5a. CONTRACT NUMBER
N/A

5b. GRANT NUMBER
N/A

5c. PROGRAM ELEMENT NUMBER
N/A

6. AUTHOR(S)
Josh L. Wilkerson, Jim Bobinchak, Michael Culp, Josh Clark,
Tyler Halpin-Chan, Katia Estabridis, Gary Hewer

5d. PROJECT NUMBER
N/A

5e. TASK NUMBER
N/A

5f. WORK UNIT NUMBER
N/A

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Air Warfare Center Weapons Division
1 Administration Circle
China Lake, California 93555-6100

8. PERFORMING ORGANIZATION REPORT NUMBER
NAWCWD TP 8786

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Naval Air Warfare Center Weapons Division (Code 400000D)
1 Administration Circle
China Lake, California 93555-6100

10. SPONSOR/MONITOR'S ACRONYM(S)
NAWCWD

11. SPONSOR/MONITOR'S REPORT NUMBER(S)
N/A

12. DISTRIBUTION/AVAILABILITY STATEMENT
DISTRIBUTION STATEMENT A: Approved for public release.

13. SUPPLEMENTARY NOTES
None.

14. ABSTRACT

(U) This report presents the two-dimensional (2D) version of the Automated Velocity Obstacle Collision Avoidance (AVOCA) system. AVOCA is a platform independent, distributed collision avoidance system for multi-agent environments. AVOCA requires minimal communication between agents, with plans to extend into sensor based agent recognition in future work. AVOCA uses well-founded velocity obstacle approaches that have been enhanced for the AVOCA system. Additionally, AVOCA uses the novel kinematic velocity obstacle (KVO) to account for agent kinematics in its calculations, also presented in this report. Results are presented for both simulations and physical experimentation, which demonstrate both the system's ability to guide agents without collision in the vast majority of cases and the effectiveness of KVOs in general.

15. SUBJECT TERMS
Automated Velocity Obstacle Collision Avoidance (AVOCA), Autonomous Operation, Collision Avoidance System, Kinematic Velocity Obstacle (KVO), Two Dimensional (2D), Unmanned Aerial Vehicle (UAV), Velocity Obstacle (VO)

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Josh L. Wilkerson
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	SAR	34	19b. TELEPHONE NUMBER *(include area code)* (760) 793-0122

CONTENTS

1.0 Introduction 3

2.0 Related Work 4

3.0 AVOCA Algorithm 5
3.1 Velocity Obstacles (VOs) 5
3.1.1 Basic and Truncated VOs 5
3.1.2 RVOs and HRVOs 9
3.2 Clearpath 11
3.3 Kinematic Velocity Obstacles (KVOs) 13

4.0 Simulation Results 16
4.1 MATLAB Simulation 16
4.2 6DOF Simulation 20
4.2.1 Graphical Components 21
4.2.2 Simulation Dynamics 21
4.2.3 Simulation Results 22

5.0 TurtleBot Experimentation Results 23
5.1 AVOCA Integration With ROS 23
5.1.1 Adaptive Monte Carlo Localization (AMCL) 23
5.1.2 Integration 24
5.1.3 Velocity Controller 24
5.1.4 Networking and Infrastructure 25
5.1.5 Test Scenario and Results 27
5.1.6 Limitations and Sources of Error 30

6.0 Conclusion 30

7.0 References 31

8.0 Nomenclature 33

Figures:

1. Example VO Construction 6
2. Positive Tangent Calculation for Convex Polygons 8
3. Comparison Between VO, RVO, and HRVO 10
4. Clearpath Example 12
5. KVO Example 15
6. Box Plots Summarizing MATLAB Experiments 18
7. Snapshot of 6DOF Simulation Arena 20
8. Multiagent 6DOF Simulation Results 22
9. Diagram of AVOCA/ROS Integration 24
10. Quantile Plots of the RTTs 26
11. Histograms of the RTTs 27
12. Summary of TurtleBot Experiment Results 29
13. Example TurtleBot Path From Experiments 29

ACKNOWLEDGMENTS

The authors wish to express appreciation for the support of the Naval Air Warfare Center Weapons Division (NAWCWD) Public Law Section 219 funds.

1.0 INTRODUCTION

Collision avoidance is one of the most important elements in autonomous operation. While many collision avoidance systems have been proposed both to detect impending midair collisions and to perform deconfliction maneuvers, it is not the intention of this technical publication to provide a comprehensive literature review of them, but rather to document a fledgling effort to develop, implement, and test collision avoidance algorithms in a small scale hardware-in-the-loop (HWIL) arena. The HWIL arena hosts four TurtleBots that serve as platforms to verify the robustness of the algorithms. The arena environment includes a real-time communication network, combined with mapping, localization, and collision avoidance algorithms. The TurtleBots come equipped with sensors and the Robot Operating System (ROS), which provided the necessary functionality for navigation (mapping and localization), allowing the authors to concentrate on communications and collision avoidance in dynamic environments.

As the number of autonomous vehicles introduced on the ground, air, and sea continues to increase for both commercial and military applications, collision avoidance is of utmost importance for safety. The U.S. military operates thousands of autonomous vehicles. Militaries from France, Israel, England, Russia, and elsewhere are also operating UxVs (unmanned vehicle, with x standing for air, ground, surface, or undersea) in ever-growing numbers. Currently, 15 of the North Atlantic Treaty Organization's (NATO's) 26 nations have unmanned systems in their inventories and the number is expected to grow (Reference 1). Future military applications will include teams of UxVs working collaboratively for intelligence, surveillance, and reconnaissance (ISR); battle space awareness; and for strike type operations. Additionally, hybrid teams that include both manned and unmanned vehicles are envisioned as part of future strategic operations.

UxVs operating autonomously either in standalone or team/swarm configuration present a challenge and is an ongoing area of research. As the level of autonomy increases, mission planning needs to transition away from the execution of specifically unrealistic deterministic scenarios to a new operational paradigm that must be established to understand and validate the decisions made in a dynamic environment. Collision avoidance algorithms are essential in order to successfully execute missions of unmanned vehicles in dynamic environments. These algorithms not only have the potential to support developmental efforts but they can also support the test and evaluation (T&E) community. Recognizing such need has guided the developmental effort presented in this report, addressing not only algorithm development and simulation but also hardware implementation.

This report presents the two-dimensional (2D) version of the Automated Velocity Obstacle Collision Avoidance (AVOCA) system, a collection of velocity obstacle (VO) based collision avoidance algorithms. The primary goal of the AVOCA system is to

achieve cooperative collision avoidance by dynamic entities in the problem space (agents) performed in a distributed fashion with minimal communication requirements. The algorithms used in AVOCA achieve implicit cooperation through their application and require only basic information (i.e., position and velocity) information on other agents for their calculations. AVOCA has been implemented both in simulation and in the HWIL arena with great success; these experiments, along with their results, are detailed in this report as well.

The remaining sections of this report are organized as follows:

- Section 2.0 discusses the research efforts most critical to AVOCA.
- Section 3.0 details the AVOCA algorithms and their implementation.
- Section 4.0 describes the simulation systems used to test AVOCA, along with simulation results.
- Section 5.0 describes the integration of AVOCA with the TurtleBot systems and the experiments conducted with the TurtleBots and their results.
- Section 6.0 summarizes the information presented.

2.0 RELATED WORK

VO collision avoidance has been used for at least the last century, with the earliest recorded usage occurring in 1903 for port navigation (Reference 2). Since then, the approach has been rediscovered a number of times, with Fiorini et al. first using the term velocity obstacle in 1993 (Reference 3). Being such a time-tested, effective approach, there is a wide variety of VO algorithms. The AVOCA system uses basic VOs (Reference 3), truncated VOs (References 4 and 5), reciprocal velocity obstacles (RVOs) (Reference 6), hybrid reciprocal velocity obstacles (HRVOs) (Reference 7), and Clearpath (Reference 5).

Generally, a VO is a geometric region (typically an infinite triangle) that is calculated using two agents in the problem space, a *source agent* (A_{src}) (i.e., the agent that is being guided by the algorithm) and an *other agent* (A_{oth}). The VO region defines the set of all points that, if used for the endpoint for A_{src}'s velocity vector, will result in a collision between the two agents at some point in the future. A characteristic of basic VOs is that there is no assumed cooperation between agents (i.e., between entities in the problem space). If the agents are actually cooperating, then this results in wide avoidance trajectories, which can be a problem in close quarters situations. RVOs, introduced by Van den Berg et al. in 2008, solve this problem by allowing agents to split the task of avoidance. However, one issue remained with VOs and RVOs, namely the possibility for reciprocal dances. Basically, this is when two agents repeatedly attempt to pass each other on the same side. The HRVO was proposed by Snape et al. in 2011 and solves this

problem by combining components from both VOs and RVOs; this action essentially removes a plane of symmetry between the two agents' VO regions. These algorithms are described in more detail in the following sections.

3.0 AVOCA ALGORITHM

An assumption is made that each agent can be represented by a convex shape. Typically, these are either circular regions (representing the agent and optionally a safety keep-out region) or convex polygons, constructed from points that represent possible agent locations. Often an agent's precise location cannot be known (Reference 8), and so a set of possible locations (typically based on sensory data) are generated. The set of points is provided to AVOCA to determine the convex hull for the point cloud provided using the monotone chain algorithm (Reference 9). Since each point in the point cloud represents a possible location for the center of the agent, AVOCA also provides functionality for the user to define a physical shape for the agent, which it will use in these calculations to create a true shape/region that will completely contain the agent.

3.1 VELOCITY OBSTACLES (VOs)

To perform its avoidance calculations, AVOCA builds basic VOs, RVOs, and HRVOs for all other agents in the problem space. The constructs are built using the velocity and position of each agent, and so these data items are required by AVOCA. Currently, AVOCA relies on inter-agent communication for this information; however, it is planned in future work to allow agents to get these data from onboard sensors. This addition will make the collision avoidance system completely localized.

3.1.1 Basic and Truncated VOs

As mentioned previously, basic VOs assume no inter-agent cooperation. When used unmodified, this means that A_{src} assumes full responsibility for performing the collision avoidance between the two agents. This strategy works well in situations where the other agent is noncooperative; however, it also results in overly conservative avoidance maneuvers when both agents are attempting to avoid each other. In its current state, AVOCA only uses unaltered VOs in its avoidance calculations when the basic VO does not indicate an impending collision between the agents. Including noncooperative agents in the AVOCA calculations is left as future work.

While there are a number of ways to approach VO construction, the method described here will perform the construction in place (i.e., in the global problem space) as much as possible in an effort to simplify the process/description. Additionally, to make some of the calculations more intuitive, the AVOCA system uses bound Euclidean vectors (i.e., vectors in which both the base and end point are used), rather than the more

commonly used free vectors (i.e., vectors in which just the magnitude and angle are relevant). Figure 1 shows a visual example of the VO construction process. This figure will be discussed as the VO construction process is described.

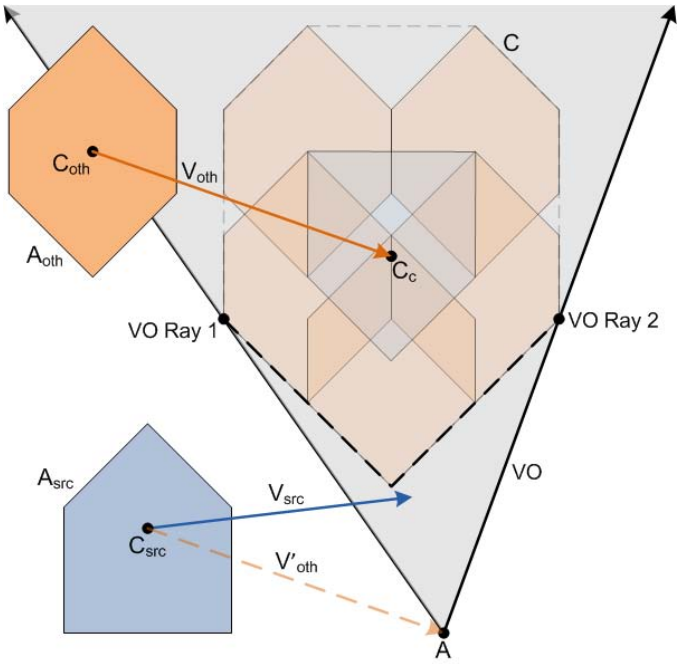

FIGURE 1. Example VO Construction.

The first step of basic VO (henceforth referred to as simply VO) construction is to find the apex of the VO region. This apex is found by transposing the velocity vector of A_{oth} (V_{oth}) to the centroid of A_{src} (C_{src}); in Figure 1, the transposed vector is labeled V'_{oth}. The endpoint of the transposed vector defines the apex of the VO region (A).

The next step is to calculate the combination of A_{src} and A_{oth}. Conceptually, in order to avoid A_{oth}, A_{src} must account for both its own shape as well as A_{oth}'s. This step is done by calculating the Minkowski sum of the two agent shapes. For circular agents, this process involves simply adding the radii of the two agents. For convex polygons, the sum is calculated by transposing each shape to the origin and adding each point in A_{oth} to all points in A_{src}. Another way of thinking of this is to take the shape for A_{oth} and copy it for each point in A_{src}, placing the center of each copy on the points in A_{oth}. After performing this sum, the new convex hull is once again calculated for the result (C).

However, consider the case where A_{src} and A_{oth} are facing each other, and A_{src} intends to pass A_{oth} by going below it (if this situation is viewed from a top-down perspective). In this case, the top of A_{src}'s shape must be able to go past the bottom of A_{oth}'s shape. But in the calculation just described, the bottom of C will be defined by the bottom of A_{src} summed with the bottom of A_{oth} (by virtue of the convexity of the shape), where it needs to be defined by the top of A_{src} summed with the bottom of A_{oth}. This can be accounted

for by rotating the shape for A_{src} by 180 degrees before doing the summation (simply achieved by negating all points in A_{src} after being transposed to the origin).

Once the shapes for A_{src} and A_{oth} are combined into C, it must then be placed relative to A. The center for C (C_c) is calculated by subtracting the center for A_{oth} (C_{oth}) from C_{src} and adding the result to A. When constructed in place, C_c will always be the endpoint of V_{oth} as drawn from C_{oth}. This process can be seen in Figure 1, where an inverted copy of A_{src} was placed at C_c, and then copies of A_{oth} are placed on each node on the A_{src} copy. The resulting convex polygon C is shown by the dashed line encompassing the result of this process.

The final step is to calculate the two rays that will define the VO region. The VO rays are defined by the two tangent lines for C that pass through A; in other words, the two lines that contain A and are as wide apart as possible while still touching C. For circular agents, the tangents are found using basic trigonometry. For convex polygon agents, the tangents are found by iterating over each point, calculating the z-component of the cross product between a vector from A to the current point and a vector from A to each other point in the hull. When a point is found that has a positive result for all other points in the hull, then that point defines one of the tangents. Similarly, when a point is found that has a negative result for all points, then that point defines the other tangent. Figure 2 shows the algorithm for this process for the positive tangent in pseudocode. The only difference in this algorithm for negative tangent calculation is that on the last line of the *if* block in the *for* loop, the z-component of R is checked to be less than or equal to 0. These points are marked in Figure 1 next to the labels for the VO rays. Once both tangents are found, the rays are set and stored, and the VO is complete.

If there is uncertainty in an agent's position, then there is a possibility that the agent shapes could overlap without a collision occurring. If this happens, then when a VO between the two agents is built, the VO apex will fall inside the C shape. Because the tangent rays must contain the entire C shape, and are based at A (which is now inside of C), attempting to calculate the rays using the procedure previously described will invariably result in error. In AVOCA, if A falls inside of C, then a special VO construction process is used with the goal of generating near-flat VO regions that encourage the agents to separate. Near-flat means that the smaller angle between the VO rays is less than π radians by an arbitrarily small amount. If the rays are exactly π radians apart, then the decision for which side of the rays the VO region falls on is determined by floating point error rather than the intention of AVOCA (as the VO region must be convex, and so is always on the side of the rays defined by the smaller angle between them).

```
polygon ← The set of convex polygon nodes
A ← The VO apex

done := false
for curNodeIdx := 0...|polygon|-1 and done = false do
    P := polygon[curNodeIdx]
    allPos := true
    for i := 0...|polygon|-1 and allPos = true do
        if i ≠ curNodeIdx then
            M := polygon[i]
            R⃗ := AP⃗ × AM⃗
            allPos := (R_z ≥ 0)
        end if
    end for
    if allPos = true then
        Ray1⃗ := AP⃗
        done := true
    end if
end for
```

FIGURE 2. Positive Tangent Calculation for Convex Polygons.

If circular agents are being used, then the VO for the overlapping agent case is constructed by drawing a vector starting at C_c and initially ending at A (which, if you recall, is inside of C). The vector magnitude is then adjusted to be the radius of C plus a small amount (enough to place the endpoint just outside of C, 0.0002 was arbitrarily selected for the experiments presented), while the angle of the vector is maintained. The endpoint of this vector is now used as the new VO apex, and the VO construction process continues on as previously described.

If convex polygons are being used to represent the agents, then the VO is constructed by first finding the point on the polygon edge that is closest to A, let this point be P (note that P can fall on both the edges and the nodes on the polygon). Next, a vector is drawn from the centroid of C to P. Just like for circular agents, the magnitude for this vector is adjusted to make the endpoint fall slightly outside of C and this endpoint is used as the new VO apex. Since C is a convex polygon, the points defining the VO rays must be the two nodes for the edge on C that contains P. If P is one of the nodes on C, then P and the two nodes adjacent to P on C are considered, with the two points resulting in the widest VO being selected.

Truncated VOs are a type of VO that is useful for static/stationary objects/agents. In AVOCA, VOs are truncated if the ratio of the slower agent speed to the faster agent speed is less than a constant threshold value (2.5% was arbitrarily selected as a proof of

concept for all experiments presented) or if either agent has a zero speed (covering the very slow speed edge case). In other words, a VO is truncated if either agent can be effectively assumed to be stopped to the other agent.

To build a truncated VO, first a normal VO is built as described previously. The VO is then truncated by terminating the VO rays at the ray defining points on *C* (rather than at *A*). The bottom of the VO is then defined by the section of line segments on *C* that are between the VO defining points while maintaining the convexity of the VO (i.e., the set of line segments that are closer to *A*). If the VO in Figure 1 was to be truncated, the bottom of the VO would be defined by the two line segments on the bottom side of *C* between the VO ray defining points; in the figure, these line segments are dashed and drawn in bold on *C* (note that this particular VO would not be truncated by AVOCA, this is just used as an example for this discussion).

3.1.2 RVOs and HRVOs

As stated in the previous section, one of the major drawbacks of basic VOs is that each agent assumes full responsibility for avoiding the other agents, regardless of the potential for collaboration from other agents. RVOs were developed to help deal with this situation. When employing RVOs, the implicit assumption is other cooperating agents will share the task of avoidance equally. In the problem space, this often results in VO regions that are smaller in the immediate area of operation, allowing for more effective use of the space.

RVOs are created by first constructing the VO, as described previously. Next, the VO apex is shifted to be the midpoint between *A* and the endpoint of V_{src}; the selection of the midpoint for this shift can be thought of as the agents each taking half of the load in avoiding each other. The resulting VO region is the RVO. One key restriction on RVO construction is that the modifications should not be made if the endpoint of V_{src} does not fall in the basic VO. If the V_{src} endpoint was not in the basic VO, then the apex for the RVO could possibly fall outside the basic VO region. This would be an inaccurate VO region, as it would then include points that would not result in a collision. This restriction guarantees that the RVO will always be a subset of the basic VO.

One of the major downsides of RVOs (and technically basic VOs as well) is the risk of entering a reciprocal dance (Reference 10). This risk occurs when the two agents are facing each other and repeatedly select the same side to pass on. This issue is addressed by HRVOs (Reference 7) and is achieved by making the modifications to the basic VO to favor the source bot's current velocity (i.e., encourage the bot to change course as little as possible). To describe the HRVO construction process, first examine what was done in the RVO construction process. Two strips were removed from the VO to make the RVO; one that followed each ray (with overlap between the two apices). HRVOs are constructed by adding one of these strips back in, to make the modification nonsymmetric (which is what will allow for the bias in the algorithm). The strip added back in is the one furthest away from the V_{src} endpoint. Another way of looking at this is that the

HRVO will be constructed using either ray 1 from the VO and ray 2 from the RVO or ray 1 from the RVO and ray 2 from the VO (with the RVO ray extended in both cases to find the HRVO apex).

Figure 3 illustrates the differences between the three VO types described. In this figure, the three apices are indicated for the three VOs shown. The VO is the same region used in Figure 1, the RVO (the darkest VO region in the figure) is constructed by shifting the VO apex to the midpoint between V_{src}'s endpoint and the VO apex, and the HRVO is constructed by adding the right strip of the VO back onto the RVO (with the left RVO ray extended along the yellow line, to get the HRVO apex).

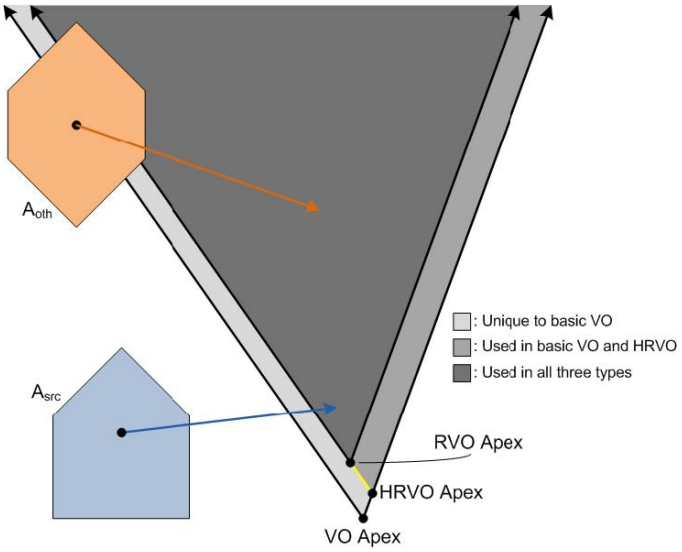

FIGURE 3. Comparison Between VO, RVO, and HRVO.

A characteristic of the VOs between two agents is that if the VO region is rotated 180 degrees about the midpoint between the agent velocity endpoints, then the resulting region is the VO from the other bot's perspective (i.e., as if A_{oth} were the source). Using this, it can be seen how the HRVO modification achieves implicit agent cooperation. Consider Figure 3. In this example, the endpoint for V_{src} is reasonably close to either ray of the RVO (recall that this is the darkest region in the figure), which is an ideal condition for reciprocal dances to occur. The HRVO is constructed by adding the right strip from the VO back to the RVO, making the HRVO the combination of the two darker regions in this image. Now it is much more likely that the source agent will elect to simply slow down some to avoid the other agent. Now imagine the HRVO being rotated around the midpoint between the two velocity endpoints to achieve the HRVO for A_{oth}. In this case, the *left* strip from the VO was added to the RVO to create the HRVO, making it most likely that A_{oth} will simply speed up a little to avoid A_{src}. And so, cooperation is achieved with no explicit communication regarding how to do so.

It is conceivable (even likely, in some applications) that the presence of other agents will preclude selecting a velocity that will cooperate with A_{oth}. In this case, the HRVO algorithm naturally will change the nature of the implicit cooperation in future time steps (switching the sides that are added back onto the RVO).

As stated previously, the AVOCA system uses VOs and HRVOs in its calculations, but only for agents that pass an initial relevancy check. There are two conditions used to determine if an agent is relevant to A_{src}:

1. A_{oth} is in front of A_{src}
2. A_{oth} is traveling in the same direction as A_{src}

The first condition is checked by determining if any portion of the other agent's shape falls in the half-plane that contains V_{src} and is defined by the line perpendicular to V_{src}. The second condition is checked by determining if the following relationship is true:

$$-\frac{\pi}{2} \leq (\angle V_{src} - \angle V_{oth}) \leq \frac{\pi}{2} \tag{1}$$

If either condition is true, then the agent is considered relevant, and VO construction continues as described. If both conditions are false, then the agent is considered nonrelevant and is ignored in the avoidance calculations. This addition can reduce the computational load considerably for each agent, especially in clustered/congested environments.

3.2 CLEARPATH

After the VOs and HRVOs have been constructed for all other agents in the problem space, the next step in AVOCA is to determine what velocity to recommend for the source agent. While the VO regions remove some possibilities for the result, there is still an infinite set of candidate velocities remaining. The Clearpath (Reference 5) algorithm provides a method for reducing this problem space to a discrete, relatively small number of candidates. In addition, the algorithm has also been proven to always have the optimal velocity choice in the set of candidates it produces, where optimal is defined as the velocity that is as close as possible to the agent's preferred velocity (Reference 5).

In the AVOCA implementation of the Clearpath algorithm, candidate velocity endpoints are generated in one of four methods:

1. A VO region apex (if the VO is not truncated)
2. Any of the nodes along the truncation polyline (if the VO is truncated)
3. The projection of preferred velocity onto an edge of the (HR)VO, including projections onto the truncation polyline

4. A point of intersection between two VO regions

The set of candidate velocities is generated by iterating over all (HR)VOs generated for the agent, calculating the candidates using each of these four methods. Any candidate that falls inside of another VO region is removed. Once all (HR)VOs have been iterated over, the candidate that is closest to the preferred velocity is selected as the result (i.e., the suggested velocity for A_{src}). If there are no valid candidates, then the published Clearpath algorithm removes the (HR)VO for the agent farthest away from the source, and repeats the process.

Figure 4 illustrates application of the Clearpath algorithm. In this figure, the blue polygon represents the source agent, the circles represent candidate velocities generated by Clearpath, the red vector represents the source agent's preferred velocity, and the green vector represents the velocity suggested by Clearpath. The agent in the center of problem space is stationary, resulting in a truncated VO, as shown. The other two agents have HRVOs drawn for them. In this figure, there is at least one Clearpath candidate velocity generated by each of the four possible methods described.

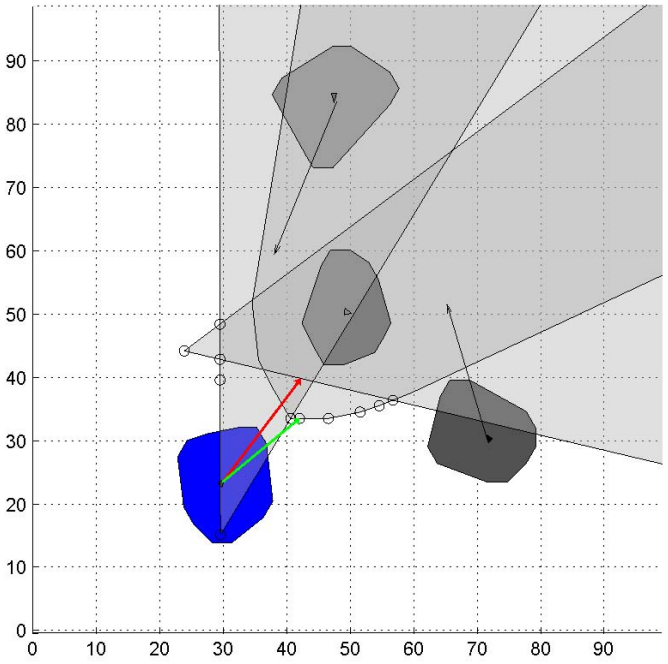

FIGURE 4. Clearpath Example.

VO removal in Clearpath is modified in the AVOCA system, as the furthest away agent, and may not be the most irrelevant agent, especially if all agents are in close quarters. If the source agent is moving, then an ignore factor is calculated for the each other agent (if the source agent is not moving then bot distance is used for this value). The first step is to rank the other agents based on distance, considering how fast the source agent is traveling:

$$rank = \left\lceil \frac{distanceToBot}{sourceSpeed} \right\rceil \tag{2}$$

This will coarsely sort the agents based on the time it will take the source agent to reach them. Next, the agents are sorted within their ranks, based on the direction they are traveling compared to the source agent:

$$ignoreFactor = rank + \left(1 - \frac{|angleDif|}{\pi}\right) \tag{3}$$

where *angleDif* is the difference between the source velocity vector angle and the other agent velocity angle on $(-\pi, \pi)$. Conceptually, this calculation makes it so that agents traveling in the same direction as the source agent will be more likely to be ignored than those traveling in the opposite direction. For example, assume there are two other agents A_1 and A_2 that are in the same rank, but A_2 is farther away from the source. Now say that A_1 is traveling in the same direction, beside the source agent, whereas A_2 is traveling directly at the source. In the original Clearpath algorithm, the VO for A_2 would be removed from consideration first, very possibly opening up the problem space, and ultimately putting the source agent on a collision course. However, if ignore factors are used, then A_2 is more important (i.e., has a lower ignore factor), and so the VO for A_1 would be removed first.

In normal operation it is quite unlikely that the set of (HR)VOs occlude the entire problem space (as that is what is required for Clearpath to not be able to generate a candidate velocity). However, the introduction of kinematic VOs (described in the next section) to the problem space make exactly the opposite true, and so this addition is largely focused on solving the side effects of their usage.

3.3 KINEMATIC VELOCITY OBSTACLES (KVOs)

KVOs are a novel concept with the goal of allowing a Clearpath based avoidance scheme that takes into account the physical capabilities of the source agent. In a general VO approach, there is nothing to stop Clearpath from suggesting a velocity that is not realistically attainable by the agent. For example, if car-like agents are being guided using Clearpath, there is nothing to stop the algorithm from suggesting an agent come to a complete stop in one time step followed by a 90-degree left turn (relative to its current heading) at 100 miles per hour. If a time step is long enough to allow the agent to get up to that speed, then that much is acceptable; but a car-like agent cannot change direction immediately, additional maneuvering is required. KVOs are intended to remedy this problem while still fitting cleanly into a VO based avoidance scheme.

KVOs are constructed based solely on the source agent's capabilities (i.e., no information from other agents is needed). They are based on (one-time) user specified

movement axes, which describe the directions and speeds an agent can travel relative to the current heading. Each movement axis has the following information:

- Angle of axis relative to current heading
- Range of radial motion on this axis
- Minimum/maximum speed for the axis
- Maximum speed increase and decrease possible in a time step on the axis (i.e., the acceleration and deceleration limits)

Clearly, some knowledge of the operating environment (i.e., time step size) is required for accurate values for the speed change data (and to a lesser degree the range of radial motion data items, though in empirical testing these are much less sensitive). However, even very rough values will help improve agent performance. If nothing else, arbitrarily large values can be used for these data items, effectively removing their role in KVO construction (the remaining data items will still provide considerable benefits).

Each movement axis has a banded velocity obstacle (BVO) constructed that is centered on the axis. The apex of the BVO is the center of the agent shape. The angle between the rays for the BVO is equal to the range of radial motion provided for the axis. If the range of motion is wider than π radians, then the BVO is split in half along the axis itself.

Next, two arcs are constructed between the BVO rays, centered at the agent center. These arcs define a band that is between the BVO rays but is not in the BVO region. The arc radii for the band are based on the agent's current speed and the speed constraints for the axis. If the agent's current velocity can be associated with the movement axis (i.e., its endpoint is in the BVO region), then the magnitude of the current velocity vector is used as the current speed (*S*) for the radii calculations. Otherwise, the agent's velocity vector is projected onto the axis, and the magnitude of the resulting vector is used for *S*. The arc radii are calculated using the following equations:

$$r_{inner} = max(S - maxSpeedDecrease, minSpeed) \tag{4}$$

$$r_{outer} = max(S + maxSpeedIncrease, maxSpeed) \tag{5}$$

If *S* is zero and at least one other axis has a non-zero *S*, then the band is not used (resulting in the BVO operating as a typical VO). This ensures that the avoidance algorithm will not attempt to direct the agent to suddenly change to travel in the opposite direction.

Lastly, normal VOs are constructed to fill in the areas between the axis BVOs. The resulting set of KVOs is then added to the set of VOs generated for the other agents and Clearpath is executed like normal. The only regions that Clearpath can use for velocity suggestions are the band(s) in the BVOs. Figure 5 shows an example agent with four

movement axes along cardinal directions (the black triangle in the bot center indicates the bot heading). The green VO regions are the BVOs and the orange regions are the VOs used to fill in between the BVOs. The forward and left (relative to the bot heading) axes of motion have active bands in their BVOs, while the backward and right bands were cleared because the velocity projection does not fall on those axes.

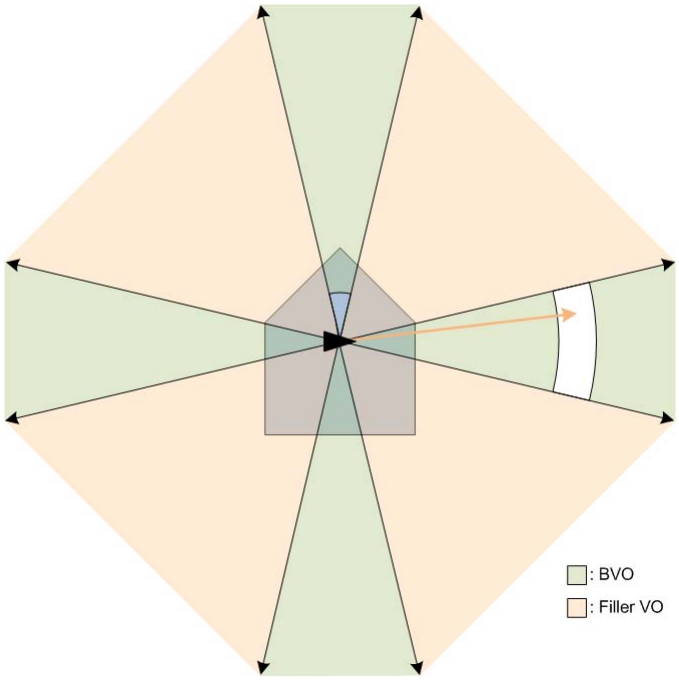

FIGURE 5. KVO Example.

As stated previously, the addition of KVOs will result in very common usage of VO removal in Clearpath, as the majority of the problem space is typically blocked by the KVOs. For example, consider Figure 5. In the situation shown, the agent only has two small regions (the band around the velocity vector, and a small region near the agent center) from which to select velocities, relative to the same problem space with no KVOs. It would be relatively easy to add a single other agent that would occlude both of these regions. If multiple other agents are added, it becomes more likely that the entire problem space is occluded than not. Additionally, in AVOCA KVOs can never be removed in Clearpath's VO removal process (they are given arbitrarily low ignore factors), which is done to ensure that the velocity constraints provided are always adhered to. If the algorithm attempts to remove a KVO, then damage control (DC) velocity selection is performed instead.

The idea of DC velocity selection is that the algorithm has detected that the agent is very close to collision, and so the suggested velocity should guide the agent away from collision as best as possible, given the agent KVO set. First, the set of candidate velocities is generated for the problem space with just the KVO set present, S_1; these

represent ideal possible velocities for the agent. Next, using the rank from the other agent ignore factors, the set of closest agents are selected along with their associated VOs. The set of candidate velocities are then generated for these VOs without the KVOs present, S_2; these represent the ideal velocities that will get the agent away from collision. Lastly, the velocity from S_1 whose endpoint is closest to that of any velocity in S_2 is selected and returned as the suggested velocity. Conceptually, this finds the physically feasible velocity that will get the agent away from collision as fast as possible.

4.0 SIMULATION RESULTS

Two simulation systems were used to test the AVOCA system, a MATLAB simulation (allowing 3 degrees of freedom [DOF]) and a 6DOF simulation. Generally speaking, the simulations manage a set of virtual agents (with associated velocity, heading, and position data) and manage the time passing in the simulated environment. Agent position and velocity information is passed to AVOCA, which returns back velocities that will result in collision free trajectories.

A key difference between the simulations is how they use the information returned from AVOCA. The MATLAB simulation provides an environment where AVOCA has complete control over the agents (i.e., the results from the AVOCA calculations are followed exactly). This provides an ideal environment for AVOCA, demonstrating best case scenarios for the system.

The 6DOF simulation passes the AVOCA results through a first-order filter before applying them to the virtual agents (described in more detail in Section 4.2). The filter acts as a smoothing function for vehicle inputs in terms of speed and turning commands. This provides a valuable testing tool, demonstrating AVOCA's performance in nonideal environments.

4.1 MATLAB SIMULATION

The MATLAB Simulation uses C++ simulation code (with AVOCA integrated) to run the simulation and create data files detailing the run. These data files are then used by MATLAB to create an animation showing what happened in the run. For example, Figure 4 is a screenshot from a MATLAB animation.

The simulation is initialized through input files indicating file locations for the simulation and agent information. The agent positions are indicated via a point cloud (which is converted by AVOCA to a convex polygon). Goal locations and movement axes (if used) are also indicated for each agent. Using this information, the simulation environment is constructed, and simulation is then executed. Each time step represents a 5% step of the simulation unit time (specific units are not needed for the purposes of this

simulation, as long as all agents (and other measured values) use the same units); for example, if the simulation unit time is an hour, then a time step is 5% of an hour, or 3 minutes. This percentage can be changed as desired by the user (5% was used for the experiments shown).

Preferred velocities are generated for each agent to be the maximum agent speed heading directly at the agent's goal. As stated previously, the AVOCA suggested velocities are used unmodified for each time step. As soon as an agent reaches its goal, it holds position. Once all agents reach their goals the simulation ends. The simulation is capped at 500 time steps (arbitrarily selected), in the event that the agents cannot reach their goal locations.

Agent data were generated for the experiments shown using a random data generation tool for the MATLAB Simulator. Agents were set to be spaced equally around a ring 100 units across. The agents themselves are indicated by a cloud of between 10 and 20 points, no more than 10 units apart. Each agent was indicated to have a physical shape representable by a circle with a radius of 3, which was combined with the agent's point cloud by AVOCA. Convex polygons were made for each agent using this information. Agent goals were indicated to be diametrically across the ring from the agent starting location. Rings of 3, 6, 9, 12, and 15 agents were generated.

Two configurations were used for each agent layout, one using KVOs and one not using them. The KVOs used were the same for all agents. Each agent was given a forward and backward movement axis, with the same data values:

- Range of radial motion: 20 degrees
- Minimum speed: 0
- Maximum speed: 25
- Maximum slow down: 5
- Maximum speed up: 10

This results in 10 unique configurations (5 variations on bot number/position, each run both with and without KVOs). Each configuration was run 30 times, yielding a total of 300 runs. For each run, a safety radius of 0.25 unit was used for the agents (i.e., agent collisions only occur if the agent shapes overlapped by more than 0.5 unit). Data were recorded for the runs indicating the number of time steps used, the number of agent collisions, and the number of invalid velocity selections (as defined by the movement axes). The recorded data are summed up in the box plots shown in Figure 6[*].

[*] Note that shorthand nomenclature is used to indicate agent configurations in this figure, where "C3" indicates the circle of 3 bots, "C6" indicates the circle of 6 bots, and so on.

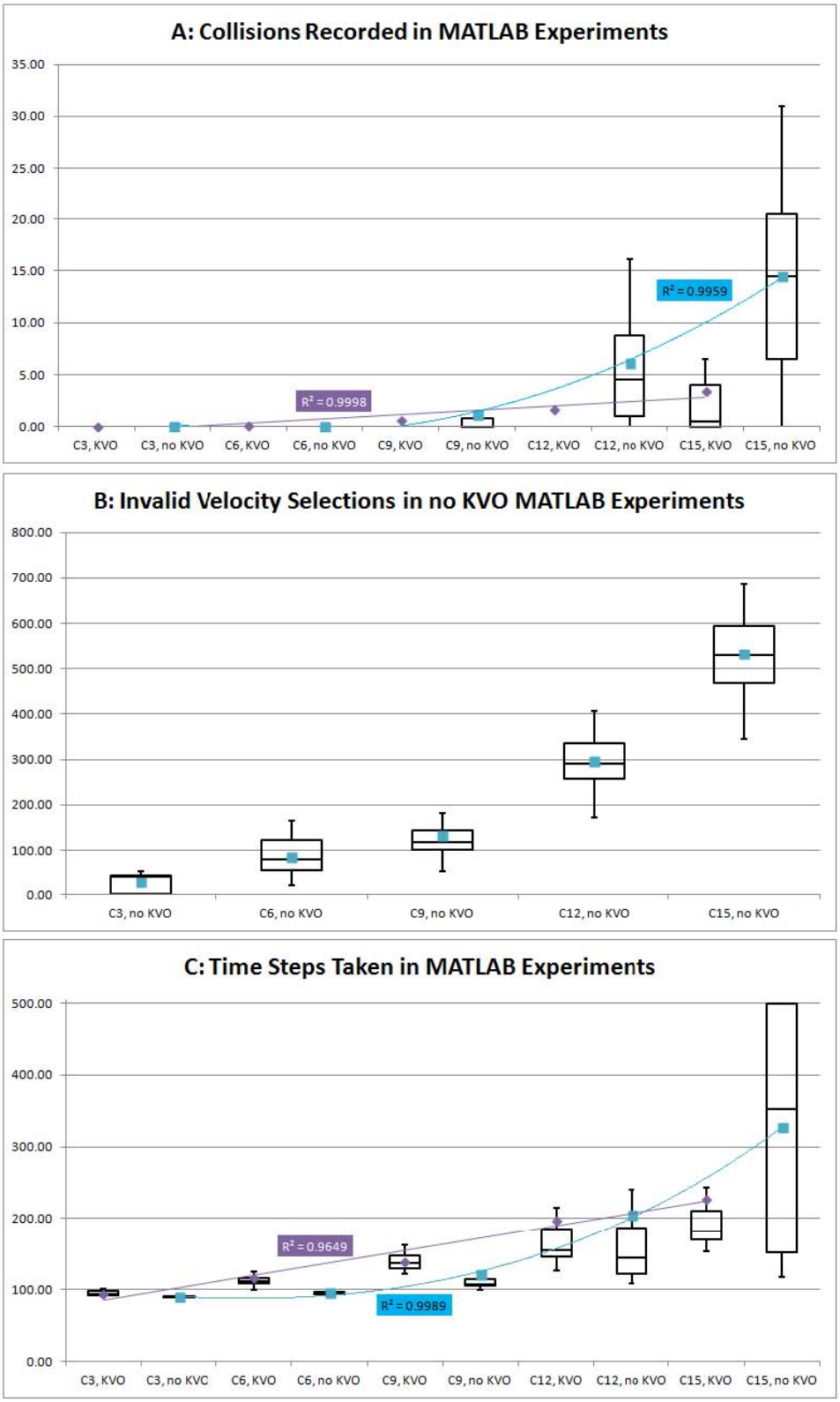

FIGURE 6. Box Plots Summarizing MATLAB Experiments.

A boxplot (Reference 11) is a simple graphical representation of a probability distribution. The edges of a box plot represent the 25th and 75th percentile with the line inside the box representing the median. Additionally, error bars indicating the 10th and 90th percentiles are included above and below each box. In Figure 6, the data averages are also shown as floating markers in the same column as the box plots.

Chart A in Figure 6 summarizes the collisions recorded during the runs. Both approaches (KVO and no KVO) did very well up to the circle of 9 experiments, recording less than 5 collisions in all runs (and 1 or less in the majority of runs). However, with 12 and 15 bots, the no KVO experiments had a notable increase in collisions; whereas the experiments using KVOs recorded less than 5 collisions in the vast majority of runs. This result is not unexpected, since without KVOs the agent movements will almost certainly be erratic when many agents are in close quarters (keeping in mind that in the MATLAB simulation, agents move exactly as AVOCA indicates).

Trendlines were also added to Chart A to visualize the growth of average collisions recorded (the floating markers in the chart); R^2 values are also shown for the trendlines (a metric indicating how closely the trendline matches the data, where 1.0 is a perfect match). For the experiments using KVOs, a linear trendline was used (shown in purple on the chart), whereas a third order polynomial was used to match the experiments that did not use KVOs (shown in cyan on the chart). This indicates that it is reasonable to expect that as the number of agents in the area of operation increases, the number of collisions should increase at a much slower rate if KVOs are used.

Chart B in Figure 6 shows the summary for the invalid velocity data recorded. As expected, the KVO experiments reported 0 invalid velocities for all experiments and so were not included in this chart. As can be seen, velocities outside of the agent's true abilities are selected very frequently when KVOs are not used, even when relatively few agents are in the area of operation. This means that when applied to real agents with physical limitations, AVOCA (and most other VO based collision avoidance algorithms) will frequently provide unrealistic velocities to the agents if KVOs are not used, which can easily be expected to result in agent collisions.

Lastly, Chart C in Figure 6 summarizes the amount of time steps taken for the experiments to complete. Recall that an experiment was ended if either all agents reached their goals or 500 time steps occurred. Generally, the KVO experiments took longer to complete for the experiments with 9 or fewer agents. This result is expected, since when KVOs are used the agents are much more restricted in their movements, resulting in (relatively) slower traversal of the problem space. When 12 agents were present, the two approaches took roughly the same amount of time to complete. However, when 15 agents were present, the experiments not using KVOs often resulted in agent deadlocks or agent scattering (i.e., an agent(s) selecting a velocity with a relatively enormous magnitude to avoid a collision, resulting in the agent moving very far away in the problem space almost instantaneously), meaning the runs were timed out at 500 time steps. When KVOs are used, agents move more predictably and realistically,

effectively sidestepping both of these issues, resulting in more predictable/stable performance.

Trendlines and R^2 values were added to Chart C in the same manner as Chart A. Once again, a linear trendline was used to match the data for experiments using the KVOs. For the experiments not using KVOs, a second order polynomial was used to match the recorded data. A similar conclusion can be drawn for Chart C as was in Chart A, namely that as the number of agents increase, the amount of time needed for the agents to traverse the area can be expected to increase at a much slower rate if KVOs are used.

4.2 6DOF SIMULATION

A graphical Multiagent 6DOF Simulation was written to test and evaluate AVOCA. The simulation allows the user to activate one of several predefined scenarios where the agents are to avoid one another. In addition, two other types of collision avoidance algorithms can be activated, namely no collision avoidance (i.e., a worst-case scenario to serve as a baseline for comparison) and force-based virtual spring collision avoidance. There is also a way to activate several graphical features used for visualization during the simulation. For example, red and yellow arrows representing the current and preferred heading of all the agents (see Figure 7) can be activated via a checkbox.

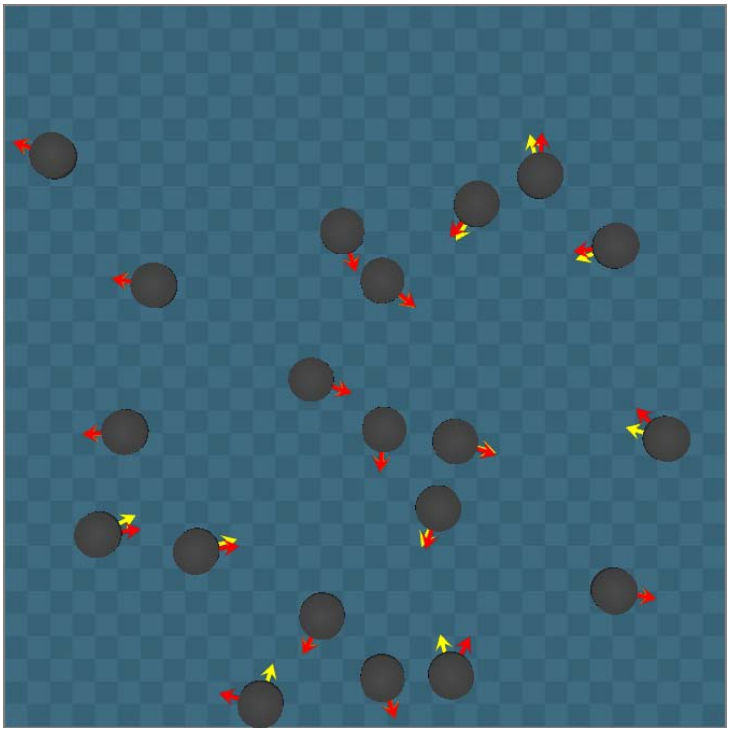

FIGURE 7. Snapshot of 6DOF Simulation Arena.

The Multiagent 6DOF Simulation currently simulates three-dimensional (3D) agents maneuvering on a 2D surface, but it can be easily extended to simulate 3D agents maneuvering in a 3D volume. This will allow testing and evaluation of 3D AVOCA algorithms currently in development for unmanned aerial vehicles (UAVs).

4.2.1 Graphical Components

Three publicly available graphics libraries are used in the Multiagent 6DOF Simulation: Open Graphics Library (OpenGL), OpenGL Utility Toolkit (GLUT), and the OpenGL User Interface (GLUI). OpenGL (Reference 12) is a cross-language, multi-platform application programming interface for rendering 2D and 3D computer graphics. The Application Programming Interface (API) is typically used to interact with a graphics processing unit to achieve hardware-accelerated rendering.

GLUT (Reference 13) is a library of utilities for OpenGL programs, which primarily perform system-level input/output (I/O) with the host operating system. Functions performed include window definition, window control, and monitoring of keyboard and mouse input. Routines for drawing a number of geometric primitives (both in solid and wireframe mode) are also provided, including cubes, spheres and the Utah teapot. GLUT also has some limited support for creating pop-up menus. The two aims of GLUT are to allow the creation of cross-platform portable code between operating systems and to make learning OpenGL easier. OpenGL programming while using GLUT often takes only a few lines of code and does not require knowledge of operating system-specific windowing APIs.

GLUI (Reference 14) is a C++ user interface library that provides controls such as push buttons, radio buttons, checkboxes, and spinners to OpenGL applications. It is window-system independent, relying on GLUT to handle all system-dependent issues, such as window, keyboard, and mouse management.

4.2.2 Simulation Dynamics

An agent's motion is governed by two ordinary differential equations:

$$\frac{\delta\, heading}{\delta t} = heading_{cmd} - heading \tag{6}$$

and

$$\frac{\delta\, speed}{\delta t} = speed_{cmd} - speed \tag{7}$$

These equations are first-order filters on the commanded input, and are solved using a second-order Adams-Bashforth numerical integrator (Reference 15). Integrating

Equation 6 (i.e., time rate of change of heading) yields the agent's heading, and integrating Equation 7 (i.e., time rate of change of speed) yields the agent's speed, and together these form the agent's preferred velocity vector. The preferred velocity vectors of all the agents are then passed to the AVOCA algorithm along with each agent's current velocity vector, and on return, each agent receives a new velocity vector that will result in collision-free travel for the current time step.

4.2.3 Simulation Results

To evaluate the performance of the AVOCA algorithm, 20 agents having a 1-foot radius were randomly positioned with random headings in a 32 x 32 square foot (ft^2) simulated arena, as shown in Figure 7. When an agent reached the arena boundary, it was given a heading that would take it back into the arena, passing near the center. Three collision avoidance cases where tested in the Multiagent 6DOF Simulation, and the results are plotted in Figure 8 as three box plots.

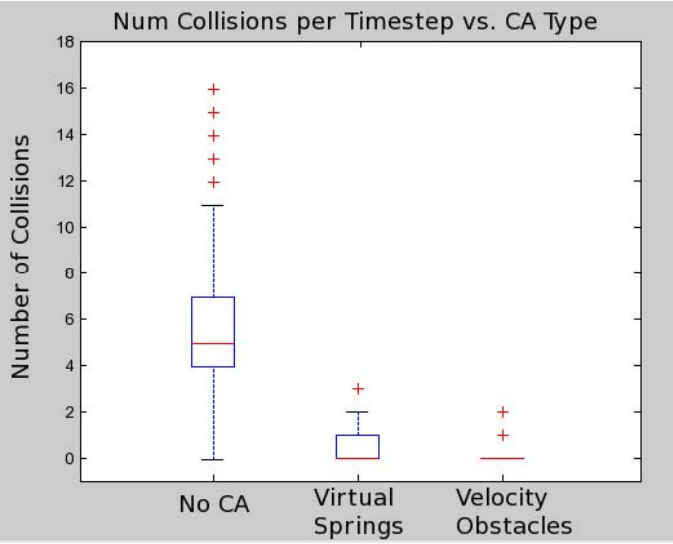

FIGURE 8. Multiagent 6DOF Simulation Results.

In the box plots show in Figure 8, all values beyond the 10th and 90th percentiles are graphed individually as a point on a graph. In this figure, the box plots for the number of agent collisions per time step that occurred during a 1-minute simulation run (each time step is 1/60 second, so 1 minute = 3,600 time steps) are shown. The leftmost boxplot in Figure 8 shows the results of the simulation when using no collision avoidance, the middle boxplot shows the results when using virtual spring coupling, and the rightmost boxplot shows the results when using AVOCA. Collision avoidance using virtual spring coupling is implemented as a repulsive spring force that activates when two or more agents enter the local neighborhood of another agent, thereby "pushing" them apart. As seen in the figure, this drastically reduced the number of collisions between agents

compared to the case when using no collision avoidance. However, the AVOCA algorithm reduces the number of collisions even further and also guarantees smooth agent trajectories.

5.0 TURTLEBOT EXPERIMENTATION RESULTS

The algorithm is implemented on four Kobuki TurtleBots. These differential-drive open robotics platforms are equipped with netbooks, Microsoft kinect sensors, Ubuntu 12.04 Precise operating system, and designed to run ROS. The netbooks use an Intel ATOM N2600 processor with four virtual cores at 1.6 gigahertz (GHz), and 1 gigabyte (GB) of random access memory (RAM). The kinect sensor is capable of capturing a depth image at a rate of 30 hertz (Hz) with a range of 0.5 to 10 meters (m). The IEEE 802.11n (2.4 GHz band) wireless network standard is used for inter-bot communication.

5.1 AVOCA INTEGRATION WITH ROS

"ROS provides libraries and tools to help software developers create robot applications. It provides hardware abstraction, device drivers, libraries, visualizers, message-passing, package management, and more. ROS is licensed under an open source, BSD license." (Reference 16)

ROS is designed to support multiple robotic platforms. ROS provides basic function libraries and hardware drivers, both of which allow implementation of the AVOCA algorithm using object oriented C++. The Internet Protocol (IP) based message passing protocol provided by ROS enables message passing between software packages without direct interaction.

5.1.1 Adaptive Monte Carlo Localization (AMCL)

AMCL is an implementation of a Monte Carlo Localization (MCL) that adds random samples to the particle map, providing a more robust localization in the absence of landmarks and in the "kidnapped robot" scenario (Reference 17). The specific implementation is credited to Brian P. Gerkey of Willow Garage who developed the ROS node "amcl," which is used in the implementation (Reference 18). After the TurtleBot is localized by the amcl node, it posts a pose array consisting of the most likely positions and orientations. The convex hull is then created and stored for a brief period of time until the AVOCA algorithm runs. The "amcl" node updates position information by resampling the pose array after observing finite linear or rotational displacements. Based on the average speed of the TurtleBots, resampling every 0.02 m or 1 degree provides a resampling rate approximately 15 Hz, based on a 0.3 meter per second (m/s) forward velocity.

5.1.2 Integration

The AVOCA algorithm was integrated into the TurtleBot and ROS through the use of multiple custom classes. The two main classes are Subscribe_And_Create_Agents and VelocityCalculator. Subscribe_And_Create_Agents instantiates all ROS publishers and subscribers, which pass data between packages, and all agents, which store the data for each TurtleBot. The VelocityCalculator uses the most recently stored data to construct all VOs and their variants then executes the modified Clearpath algorithm, which chooses the collision free velocity. This velocity is then passed to a velocity controller which issues commands to the TurtleBot. Figure 9 outlines this class communication.

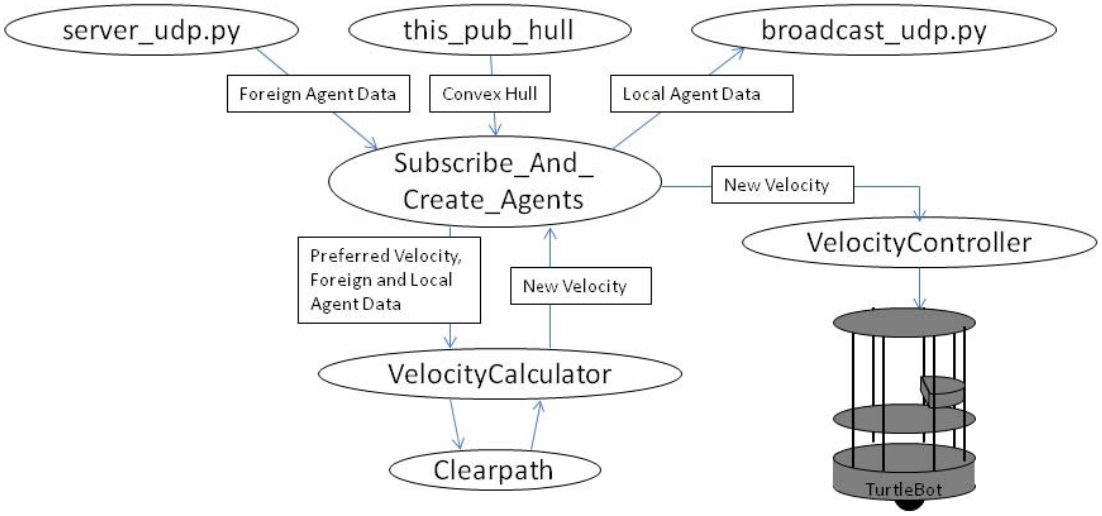

FIGURE 9. Diagram of AVOCA/ROS Integration.

5.1.3 Velocity Controller

The AVOCA code is designed around absolute velocity space. It is based on the assumption that each robot is able to immediately move in the intended direction and at the intended speed on the *xy*-plane. This assumption is unrealistic and cannot be upheld by any physical system. However, it is possible to obtain the new velocity before the next time step. Within AVOCA, KVOs ensure that selected velocities are within the limits of the physical system. The velocity controller converts these velocities to TurtleBot commands.

Due to the number of software layers between individual wheel speeds and the AVOCA algorithm, it was impractical to create a firmware level proportional integral derivative (PID) controller. Instead, an application-level proportional control was implemented. The velocity controller operates at the same rate as the AVOCA algorithm: $R_a = 5$ Hz. The bots heading and velocity must match the commanded values, θ and V,

by $1/R_a$ seconds. Therefore, the angular velocity that is issued to the bot is $V_{ang} = (\theta_g - \theta_c)/\gamma$ where θ_g is the goal heading and θ_c is the current heading, and γ is a constant representing the time allowed to complete the rotation. In the ideal case $\gamma = 1/R_a$. The linear velocity issued to the bot $V_{lin} = |V|$ where $|V|$ is the magnitude of the commanded velocity. The TurtleBot firmware handles the combination of the linear and angular velocities and conversion to individual wheel speeds. It attempts to set the speed of the left and right wheels, v_l and v_r respectively, such that $v_l + v_r = 2(V_{lin})$ and $v_l - v_r = L(V_{ang})$ where L is the distance between the wheels. In the event that the v_l or v_r are greater than the maximum wheel speed v_{max}, then the speeds are linearly reduced: for $v_l > v_r$: $v_{l,adj} = v_{max}$ and $v_{r,adj} = v_r - (v_l - v_{max})$. The $v_r > v_l$ case similarly adjusted, meaning that the bot will reduce v_{lin} in order to maintain v_{ang}. In the event that $v_l > v_r$ and $v_l > v_{max}$ and $v_l - v_r > v_{max}$ then v_l and v_r are adjusted such that $v_{l,adj} = -v_{r,adj} = (v_l - v_r)/2$, ensuring that v_{ang} is maintained at the expense of v_{lin}.

5.1.4 Networking and Infrastructure

Without the use of sensing techniques, each TurtleBot must communicate its position and velocity with the other TurtleBots. The network and communications work is basic. A course in networking and communications would cover the basic concepts that follow. An IEEE 802.11n wireless network is used for the communications. The 802.11n wireless network offers a large throughput and can work on both the 2.4 and 5 GHz frequency bands. The high throughput is not necessary for collision avoidance. Collision avoidance uses a large number packets but a small amount of data (i.e., there are many messages to send, but the messages themselves are small). Thus, latency of the network is a bigger concern than bandwidth. The different frequency bands offer a capability of swapping frequencies in case there is a source of radio interference on either of the bands. Most modern day routers, switches, or network cards have 802.11n capabilities. Additional information can be found in Reference 19.

The IP is the standard method of communication between computers via packets (Reference 20). There are two main methods for the transportation layer of IP: Transmission Control Protocol (TCP) and User Datagram Protocol (UDP). The TCP gives a guarantee of flow control, discards duplicate packets, and provides a mechanism for congestion control. The TCP is useful for applications that need reliability and correctness such as web pages or databases. The reliability given by TCP comes with an overhead cost and is less suited for real-time applications. The UDP has no reliability constraints and does read packets that are out of order and that are duplicate. The UDP has reduced overhead, making it more ideal for real-time applications that stream data or pass simple communication. For a program to use UDP for its communication, it needs to be robust against lost and out of order packets.

To implement VOs, the TurtleBots must update their velocities and positions at or faster than the VOs calculations. With AMCL, the position frequency depends on change in distance or change in angle. If using an overhead camera to track the bots, the frequency can be increased to the frame rate of the camera. Using TCP is inappropriate

for communications. If a series of packets lag, then TCP will block until it receives all packets in the correct order. The VOs will then rely on old data to calculate a new set of VOs. The UDP can continue accepting the new packets about positions and velocities when receiving packets out of order. With UDP, old and duplicated packets are dropped, leaving only the newest data for use.

A ping test was run from the four TurtleBot netbooks simultaneously to a single computer at rate of 20, 40, and 80 Hz. The round trip time (RTT) medians were 9.69, 15.72, and 16.07 milliseconds (ms) for the 20, 40, and 80 Hz tests, respectively. As the frequency increases, the latency on the RTTs increases as shown in Figure 10.

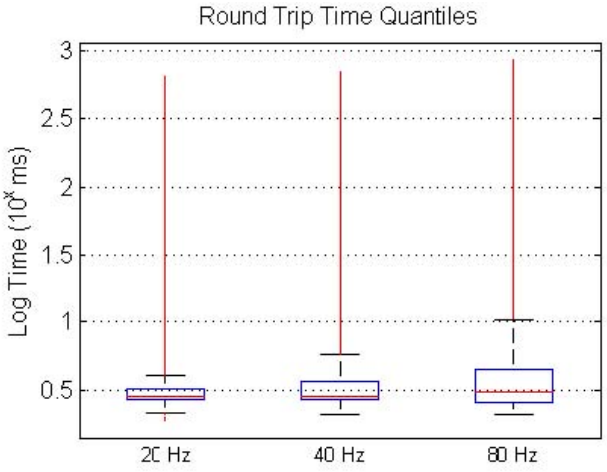

FIGURE 10. Quantile Plots of the RTTs.
The test size for each transmission rate was 40,000.

Figure 10 appears to be alarming in the number of outliers, however the number of data points used for these tests are large enough to create such a large number of outliers. In Figure 11, the RTTs appear to form an exponential distribution. The outliers of Figure 10 are in the tail and have a relatively small probability. For up to around 40 Hz the majority of RTTs are smaller than the transmission rate. With such low latencies, the position can be updated at a faster rate that the VOs and positions are calculated. This allows for extrapolation of the position with a higher transmission rate of velocities.

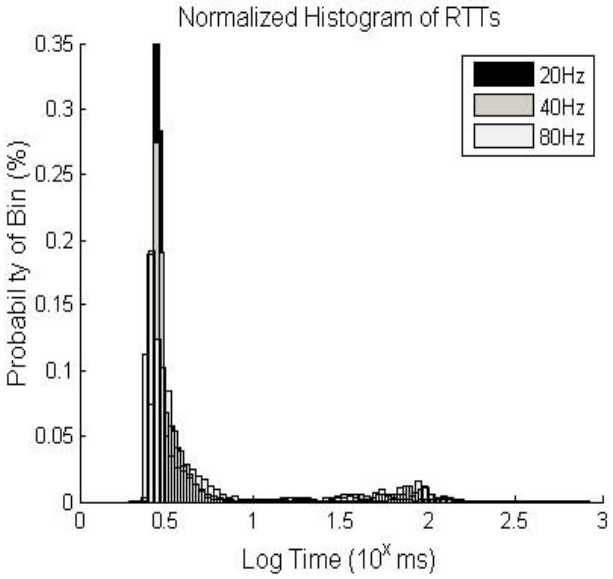

FIGURE 11. Histograms of the RTTs.

5.1.5 Test Scenario and Results

Square Scenario. The four TurtleBots are placed on four corners of a 10 ft x 10 ft square located within the arena. After performing a rotate in place operation to help with localization, each TurtleBot is instructed to move to the opposing corner. Each bot starts movement at roughly the same time; there is occasionally a noticeable delay in start times due to delayed packet transmission and individual netbook performance. The scenario was performed 30 times each for 5 different set of parameters. The following notable parameters were common to all scenarios: AVOCA Rate = 5 Hz, AMCL "Rate" = 0.02 m/1 degree. Two movement axes were used for each of the bots, a forward and a reverse. The forward axis used the following parameters: maximum speed = 0.3 m/s, minimum speed = 0.0 m/s, maximum acceleration = 0.05 meter per second squared (m/s^2), maximum deceleration = 0.05 m/s^2, and range of motion = 15 degrees. The reverse axis parameters were identical with the following exception: maximum speed = 0.2 m/s.

Each bot has a radius value *r* that represents its physical size. This value is applied to the convex hull and represents the entire area containing the bot in real space. Increasing this value provides a buffer between bots and accommodates localization errors. Additionally, it was necessary to increase *r* beyond the actual 0.17-m radius of the bots in order to mitigate error introduced in bot position from the use of delayed data, differential drive dynamics during a curved turn, and for the protection of hardware assets via a buffer region. If the bot radius is too low, then the bots will always scrape or collide (AVOCA will not waste any problem space unnecessarily, often providing velocities placing agent shapes flush with each other); if the bot radius is too high, then

the bots will never collide, but velocity space will become unnecessarily obstructed and agents may not be able to traverse narrow corridors, if that is part of the problem space.

The scenario was run 30 times for each of the following cases:

- Case A0: r =0.25 γ = 0.25 with KVOs
- Case A1: r =0.25 γ = 0.25 without KVOs
- Case B0: r =0.32 γ = 0.25 with KVOs
- Case B1: r =0.32 γ = 0.25 without KVOs
- Case C: r =0.32 γ = 0.2 with KVOs

During initial experimentation γ =0.2 was used with and without KVOs but it was found that in cases without KVOs the bots frequently overshot the desired heading and began oscillation. γ =0.25 was found to provide adequate rotational response in both the KVO and non-KVO scenarios. Figure 12 shows the results in terms of scrapes, collisions, and runs completed with no collisions, and Figure 13 shows an example of the bot paths from one of the C scenarios. Scrapes refer to instances where bots attempt to avoid, make contact, then continue forward on roughly their intended path. Collisions are characterized by bots that, after making contact, were not able to move forward and were therefore stopped to prevent hardware damage. The data show that the results are quite similar for KVOs and no KVOs for each of the bot radius values. During test scenarios it was noted that when KVOs were used the bot motion was smoother, and there was significantly less rotating in place, sudden stops, and abrupt accelerations. The presence of scrapes and collisions in the laboratory tests may be indicative of several issues: the bots are incapable of avoidance because they too often chose velocities that were within a VO, the bots are incapable of performing the commanded velocities adequately, or there is too much error in the bot position data. The increase in collisions when using a smaller bot radius suggests that the error in position is the most significant. These results agree with and reinforce the conclusions from the MATLAB experiments; namely that for small numbers of agents, the use of KVOs will result in similar performance as when no KVOs are used, but provides significantly smoother, less erratic agent behavior.

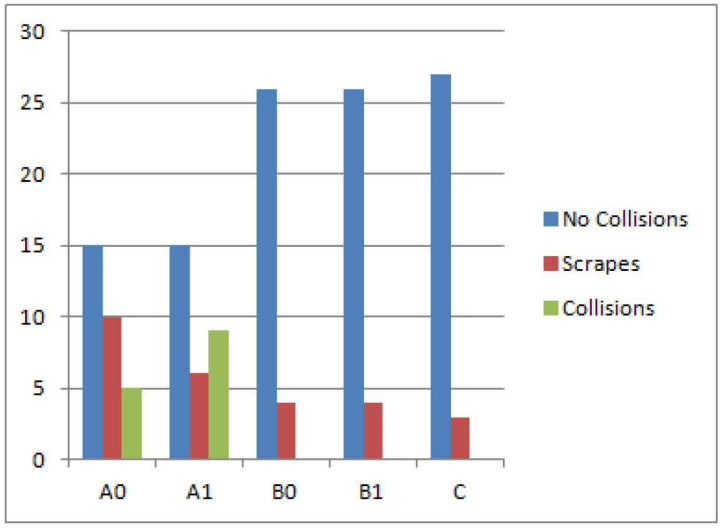

FIGURE 12. Summary of TurtleBot Experiment Results.

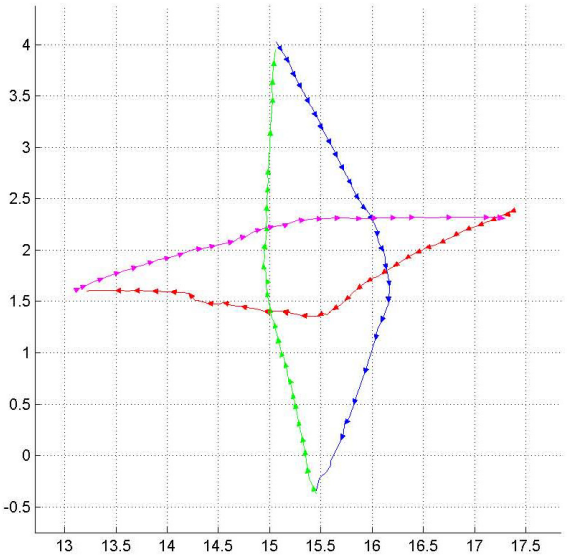

FIGURE 13. Example TurtleBot Path From Experiments.

5.1.6 Limitations and Sources of Error

Unlike nonstochastic simulations, there are multiple sources of error in every physical implementation. The following sources of error were identified and either compensated for, minimized, or noted.

- **Localization Error.** Due to the TurtleBot's configurations, when multiple TurtleBots approach each other head on, each bot obstructs the field of view of the other bot's kinect sensor, decreasing the accuracy of that bots localization. Additionally, there is the possibility of wheel slip, orientation errors caused by hardware limitations (accelerometer noise), and the effect of moving landmarks (the other TurtleBots) that decreases the localization accuracy. All of these errors are dealt with by AMCL with the effective result of increased hull size.
- **Delayed Data.** Due to the asynchronous operation of all TurtleBots, each bot will always receive out of date position data from the other bots and itself. Due to the low speeds and small distances used in this implementation, the errors in position and velocity are relatively small, and an inflated hull size is used to compensate for the error. However, in higher speed, critical or close quarters applications, the position data should be extrapolated to reflect the movement of the agents between the times when the data was taken and when the VOs are calculated.
- **Velocity Command vs. Response.** It is essential that all agents are capable of performing the commanded velocities, without this capability agent collisions are inevitable. Since the TurtleBots have a limited means of confirming that their actions have had the intended response, KVOs are used to ensure that the AVOCA algorithm issues only commands that can be performed.

6.0 CONCLUSION

Unmanned vehicles have a wide area of application that is still growing. Collision avoidance is arguably one of the most important problems for autonomous operation of these vehicles, especially as they become more prevalent. This report details the AVOCA, a distributed, limited-communication collision avoidance system. AVOCA uses VOs to perform collision avoidance, an approach that has been used in general for at least a century. Specifically, the system uses VOs, RVOs, HRVOs, KVOs (a novel contribution), and an enhanced Clearpath algorithm. The 2D AVOCA system has been tested in simulation and under physical experiments using TurtleBot systems. Results for these experiments show that the AVOCA system is able to guide agents without collision in the vast majority of cases.

7.0 REFERENCES

1. P. L. Franchi. "Near Misses Between UAVs and Airliners Prompt NATO Low-level Rules Review," *Flight International*, 2006.

2. F. S. Miller and A. F. Everett. *Instructions for the Use of Martins Mooring Board and Battenbergs Course Indicator*, Authority of the Lords of Commissioners of the Admiralty, 1903.

3. P. Fiorini and Z. Shiller. "Motion Planning in Dynamic Environments Using the Relative Velocity Paradigm," in *Proceedings of IEEE International Conference on Robotics and Automation*, 1993, pp. 560–565.

4. P. Fiorini and Z. Shillert. "Motion Planning in Dynamic Environments Using Velocity Obstacles," *International Journal of Robotics Research*, vol. 17, 1998, pp. 760–772.

5. S. J. Guy, J. Chhugani, C. Kim, N. Satish, M. C. Lin, D. Manocha, and P. Dubey. "Clearpath: Highly Parallel Collision Avoidance for Multiagent Simulation," in *ACM SIGGRAPH/EUROGRAPHICS Symposium on Computer Animation*, 2009.

6. J. van den Berg, M. C. Lin, and D. Manocha. "Reciprocal Velocity Obstacles for Real-Time Multi-Agent Navigation," in *IEEE International Conference on Robotics and Automation*, 2008.

7. J. Snape, J. van den Berg, S. J. Guy, and D. Manocha. "The Hybrid Reciprocal Velocity Obstacle," *IEEE Transactions on Robotics*, vol. 27, 2011, pp. 696–706.

8. I. J. Cox. "Blanche-An Experiment in Guidance and Navigation of an Autonomous Robot Vehicle," *IEEE Transactions on Robotics and Automation*, vol. 7, 1991, pp. 193–204.

9. A. M. Andrew. "Another Efficient Algorithm for Convex Hulls in Two Dimensions," *Info. Proc. Letters 9*, 1979, pp. 216–219.

10. Univ. Tokyo, Japan. *Simulating the Collision Avoidance Behavior of Pedestrians*, by F. Feurtey. Dept. of Electrical Engineering, Tokyo, Japan, 2000. (Masters thesis.)

11. Wikipedia. "Box Plot—Wikipedia, The Free Encyclopedia," http://en.wikipedia.org/wiki/Box plot, 2013.

12. "OpenGL Online Documentation," http://www.opengl.org/wiki/Main Page, 2012.

13. Wikipedia. "OpenGL Utility Toolkit—Wikipedia, The Free Encyclopedia," http://en.wikipedia.org/wiki/OpenGL Utility Toolkit, 2013.

14. "GLUI Online Documentation," http://glui.sourceforge.net, 2006.

15. Wikipedia. "Linear Multistep Method—Wikipedia, The Free Encyclopedia," http://en.wikipedia.org/wiki/Linear multistep method, 2013.

16. E. Fernandez. "Robotic Operating System Documentation," http://wiki. ros.org, 2013.

17. S. Thrun, W. Burgard, and D. Fox. *Probabilistic Robotics*, ser. Intelligent Robotics and Autonomous Agents Series. Mit Press, 2005. [Online]. Available: http://books.google.com/books?id=k yOQgAACAAJ

18. S. Osentoski. "Robotic Operating System Documentation amcl, http://wiki.ros.org/amcl," http://wiki.ros.org/amcl, 2013.

19. Wikipedia. "IEEE 802.11—Wikipedia, The Free Encyclopedia," http://en.wikipedia.org/w/index.php?title=IEEE 802.11&oldid=583511166, 2013. [Online; accessed 2 December 2013].

20. Wikipedia. "Internet Protocol Suite—Wikipedia, The Free Encyclopedia," http://en.wikipedia.org/w/index.php?title=Internet protocol suite&oldid=583183806, 2013, [Online; accessed 2 December 2013].

8.0 NOMENCLATURE

2D	two-dimensional
3D	three-dimensional
AMCL	Adaptive Monte Carlo Localization
A_{oth}	other agent
API	Application Programming Interface
A_{src}	source agent
AVOCA	Adaptive Velocity Obstacle Collision Avoidance [system]
BVO	banded velocity obstacle
C_{oth}	center for A_{oth}
C_{src}	centroid of A_{src}
DC	damage control
DOF	degrees of freedom
ft^2	square foot
GB	gigabyte
GHz	gigahertz
GLUI	OpenGL User Interface
GLUT	OpenGL Utility Toolkit
HRVO	hybrid reciprocal velocity obstacle
HWIL	hardware-in-the-loop
Hz	hertz
I/O	input/output
IP	Internet Protocol
ISR	intelligence, surveillance, and reconnaissance
KVO	kinematic velocity obstacle
m	meter
m/s	meter per second
m/s^2	meter per second squared
MCL	Monte Carlo Localization
ms	millisecond

NATO	North Atlantic Treaty Organization
NAWCWD	Naval Air Warfare Center Weapons Division
OpenGL	Open Graphics Library
PID	proportional integral derivative
RAM	random access memory
ROS	Robot Operating System
RTT	round trip time
RVO	reciprocal velocity obstacle
S	speed
T&E	test and evaluation
TCP	Transmission Control Protocol
UAV	unmanned aerial vehicle
UDP	User Datagram Protocol
UxV	unmanned vehicle, with x standing for air, ground, surface or undersea
V_{oth}	velocity vector of A_{oth}
VO	velocity obstacle

INITIAL DISTRIBUTION

1 Defense Technical Information Center, Fort Belvoir, VA
4 University of Utah, Salt Lake City, UT
  Bareiss, D. (2)
  Van den Berg, J. (2)

ON-SITE DISTRIBUTION

4 Code 400000D
  Boyd, M. (2)
  MacArthur, S. (2)
2 Code 4L0000D, Merwin, L.
2 Code 4L4000D, Hewer, G.
8 Code 4L4100D
  Estabridis, K. (2)
  Pentony, J. (2)
  Peterson, L. (2)
  Tyler, K. (2)
2 Code 4L6100D (Scientific and Technical Library Branch, archive copies)
2 Code 4L6200D (Technical Communication Office, file copies)
2 Code 452000D, Kirchner, R.
2 Code 47H000D, McCauley, H.
4 Code 471200D
  Bobinchak, J. (2)
  Clark, J. (2)
4 Code 471800D
  Culp, M. (2)
  Wilkerson, J. L. (2)
2 Code 472100D, Halpin-Chan, T.
2 Code 476000D, Schultz, R.